



OpenMP* Past, Present and Future

Tim Mattson

Intel Corporation

Microprocessor Technology Labs

timothy.g.mattson@intel.com

OpenMP* Overview:

`C$OMP FLUSH`

`#pragma omp critical`

`C$OMP THREADPRIVATE(/ABC/)`

`CALL OMP SET NUM THREADS(10)`

OpenMP: An API for Writing Multithreaded Applications

- A set of compiler directives and library routines for parallel application programmers
- Greatly simplifies writing multi-threaded programs in Fortran, C and C++
- Standardizes last 20 years of SMP practice

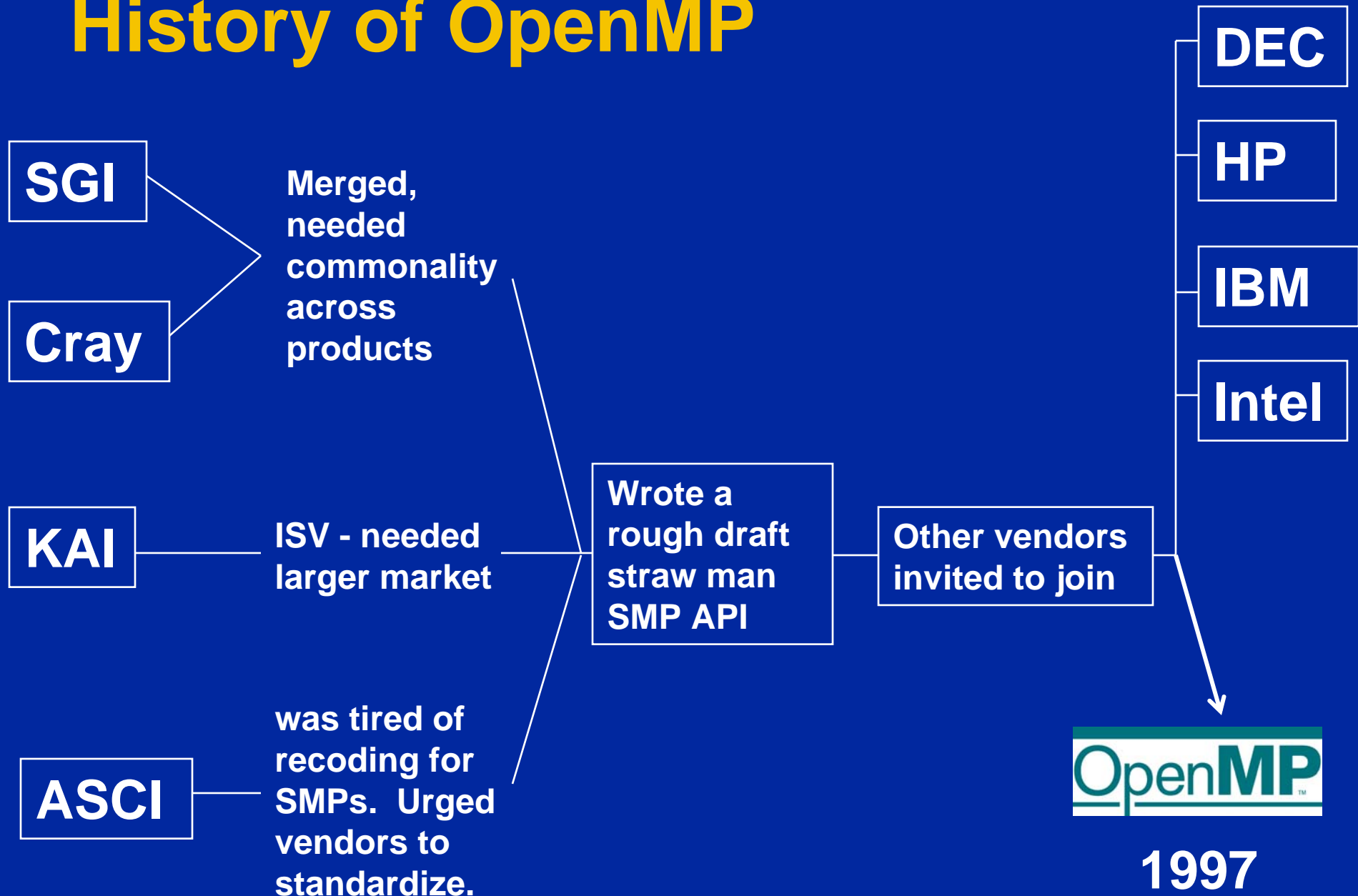
`C$OMP PARALLEL COPYIN(/blk/)`

`C$OMP DO lastprivate(XX)`

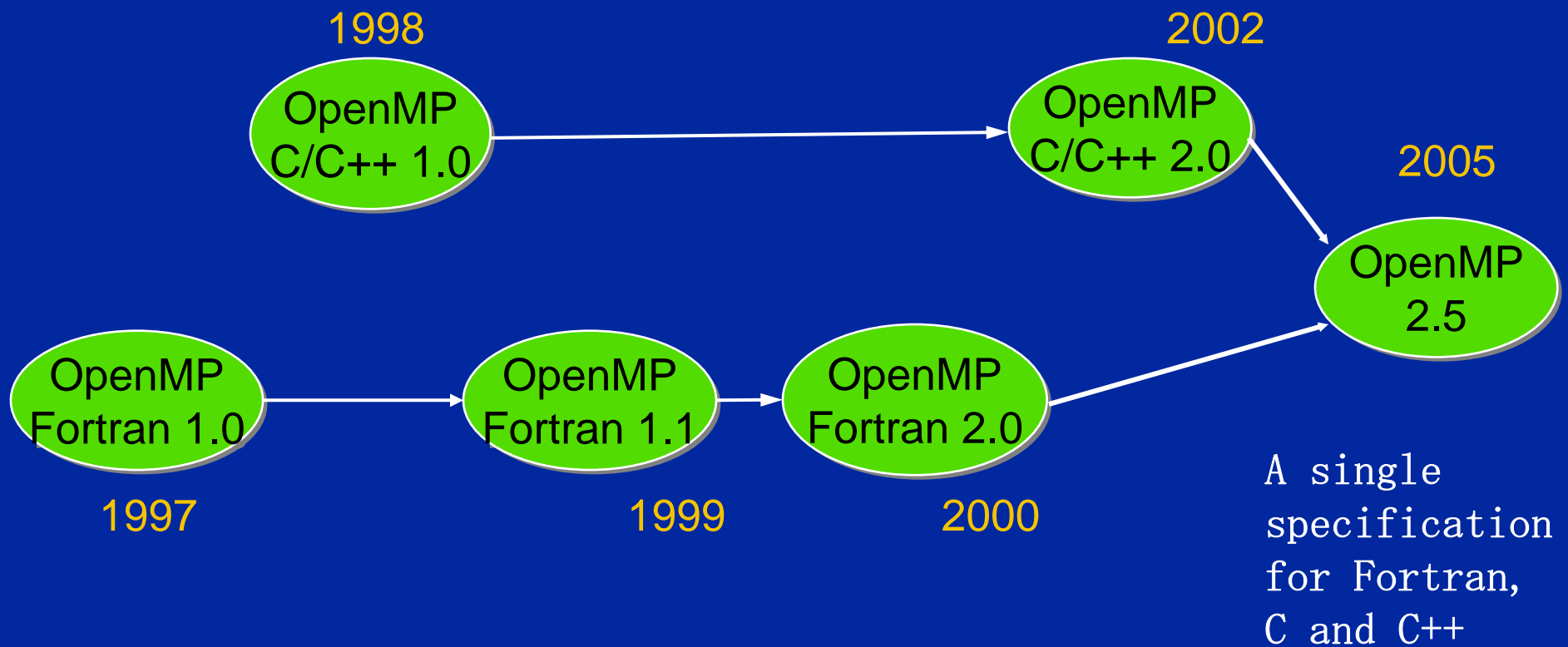
`Nthrds = OMP_GET_NUM_PROCS()`

`omp_set_lock(lck)`

History of OpenMP



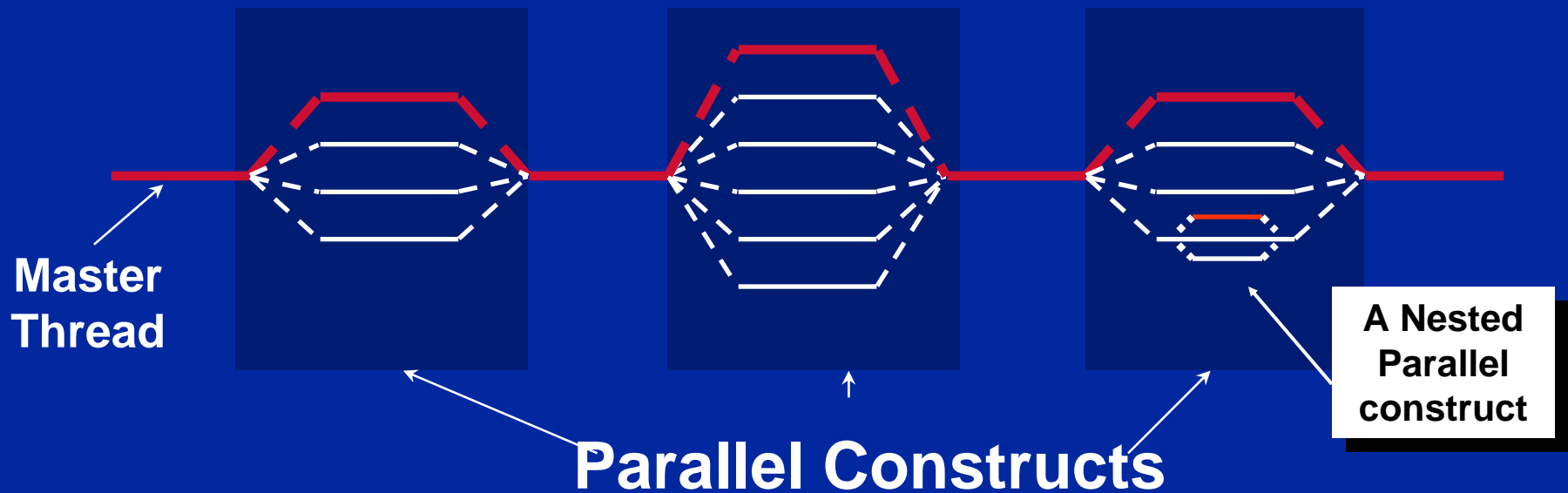
OpenMP Release History



OpenMP Programming Model:

Fork-Join Parallelism:

- ◆ Master thread spawns a team of threads as needed.
- ◆ Parallelism is added incrementally until desired performance is achieved: i.e. the sequential program evolves into a parallel program.



OpenMP loves simple loops:

The sequential “PI” program

```
#include <omp.h>
static long num_steps = 100000;    double step;
void main ()
{
    int i;  double x, pi, sum = 0.0;

    step = 1.0/(double) num_steps;

    #pragma omp parallel for reduction(+:sum) private(x)
    for (i=1;i<= num_steps; i++){
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
}
```

PI Program: Win32 API

```
#include <windows.h>
#define NUM_THREADS 2
HANDLE thread_handles[NUM_THREADS];
CRITICAL_SECTION hUpdateMutex;
static long num_steps = 100000;
double step;
double global_sum = 0.0;

void Pi (void *arg)
{
    int i, start;
    double x, sum = 0.0;

    start = *(int *) arg;
    step = 1.0/(double) num_steps;

    for (i=start;i<= num_steps;
i=i+NUM_THREADS) {
        x = (i-0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    EnterCriticalSection(&hUpdateMutex);
    global_sum += sum;
    LeaveCriticalSection(&hUpdateMutex);
}
```

```
void main ()
{
    double pi; int i;
    DWORD threadID;
    int threadArg[NUM_THREADS];

    for(i=0; i<NUM_THREADS; i++)    threadArg[i] = i+1;

    InitializeCriticalSection(&hUpdateMutex);

    for (i=0; i<NUM_THREADS; i++) {
        thread_handles[i] = CreateThread(0, 0,
            (LPTHREAD_START_ROUTINE) Pi,
            &threadArg[i], 0, &threadID);
    }

    WaitForMultipleObjects(NUM_THREADS,
        thread_handles, TRUE, INFINITE);

    pi = global_sum * step;

    printf(" pi is %f \n",pi);
}
```

OpenMP with “complicated” loops

OpenMP can't handle pointer following loops

```
nodeptr list, p;  
for (p=list; p!=NULL; p=p->next)  
    process(p->data);
```

Intel has proposed (and implemented) a taskq construct to deal with this case:

```
nodeptr list, p;  
  
#pragma omp parallel taskq  
for (p=list; p!=NULL; p=p->next)  
#pragma omp task  
    process(p->data);
```


Task queue example - FLAME: Shared Memory Parallelism in Dense Linear Algebra

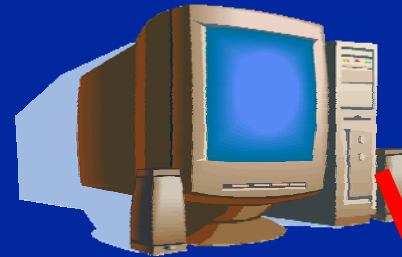
- Traditional approach to parallel linear algebra:
 - ◆ Only parallelism from multithreaded BLAS
- Observation:
 - ◆ Better speedup if parallelism is exposed at a higher level
- FLAME approach:
 - ◆ OpenMP task queues

Tze Meng Low, Kent Milfeld, Robert van de Geijn, and Field Van Zee. "Parallelizing FLAME Code with OpenMP Task Queues." *TOMS*, submitted.

Developing Libraries

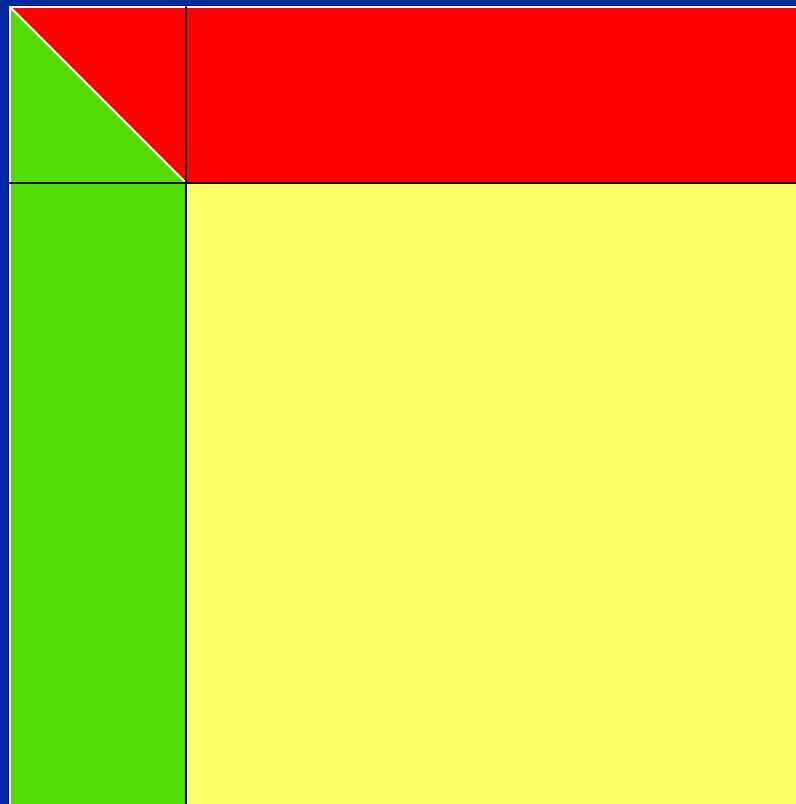
- Traditional approach: Expert painstakingly develops and implements an algorithm

operation

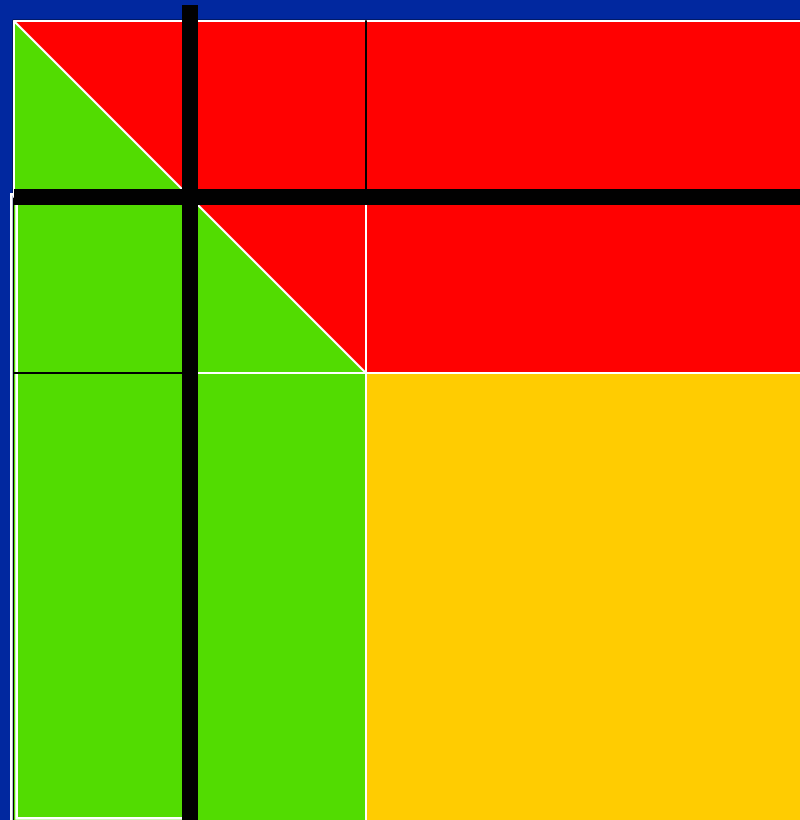


```
*      Use unblocked code
*
*      CALL DTRTIZ( UPLO, DIAG, N, A, LDA, INFO )
*      ELSE
*
*      Use blocked code
*
*      IF( UPPER ) THEN
*
*      Compute inverse of upper triangular matrix
*
*      DO 20 J = 1, N, NB
*          JB = MIN( NB, N-J+1 )
*
*      Compute rows 1:j-1 of current block column
*
*      CALL DTRMM( 'Left', 'Upper', 'No transpose', DIAG, J-1,
*      $          JB, ONE, A, LDA, A( 1, J ), LDA )
*      CALL DTRSM( 'Right', 'Upper', 'No transpose', DIAG, J-1,
*      $          JB, -ONE, A( J, J ), LDA, A( 1, J ), LDA )
*
*      Compute inverse of current diagonal block
*
*      CALL DTRTIZ( 'Upper', DIAG, JB, A( J, J ), LDA, INFO )
*      20 CONTINUE
*      ELSE
```

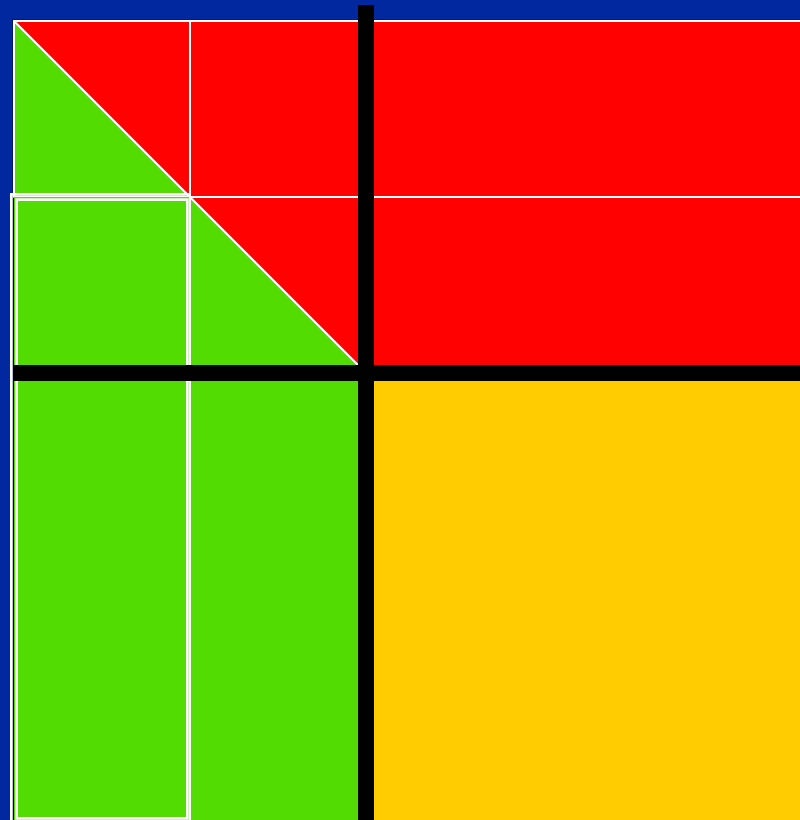
A Typical Linear Algebra Algorithm (LU factorization)



A Typical Linear Algebra Algorithm (LU factorization)



A Typical Linear Algebra Algorithm (LU factorization)

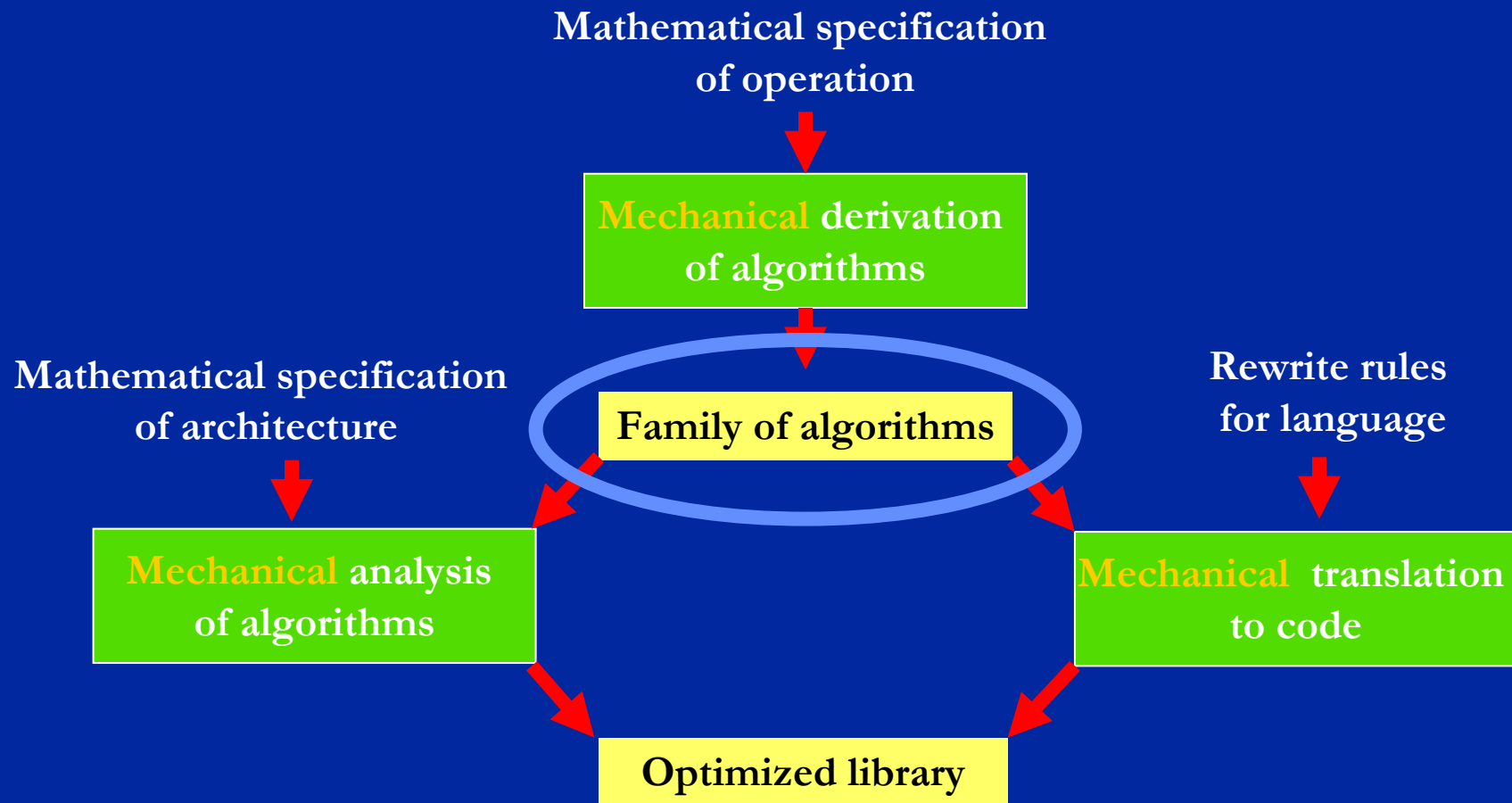


Notation:

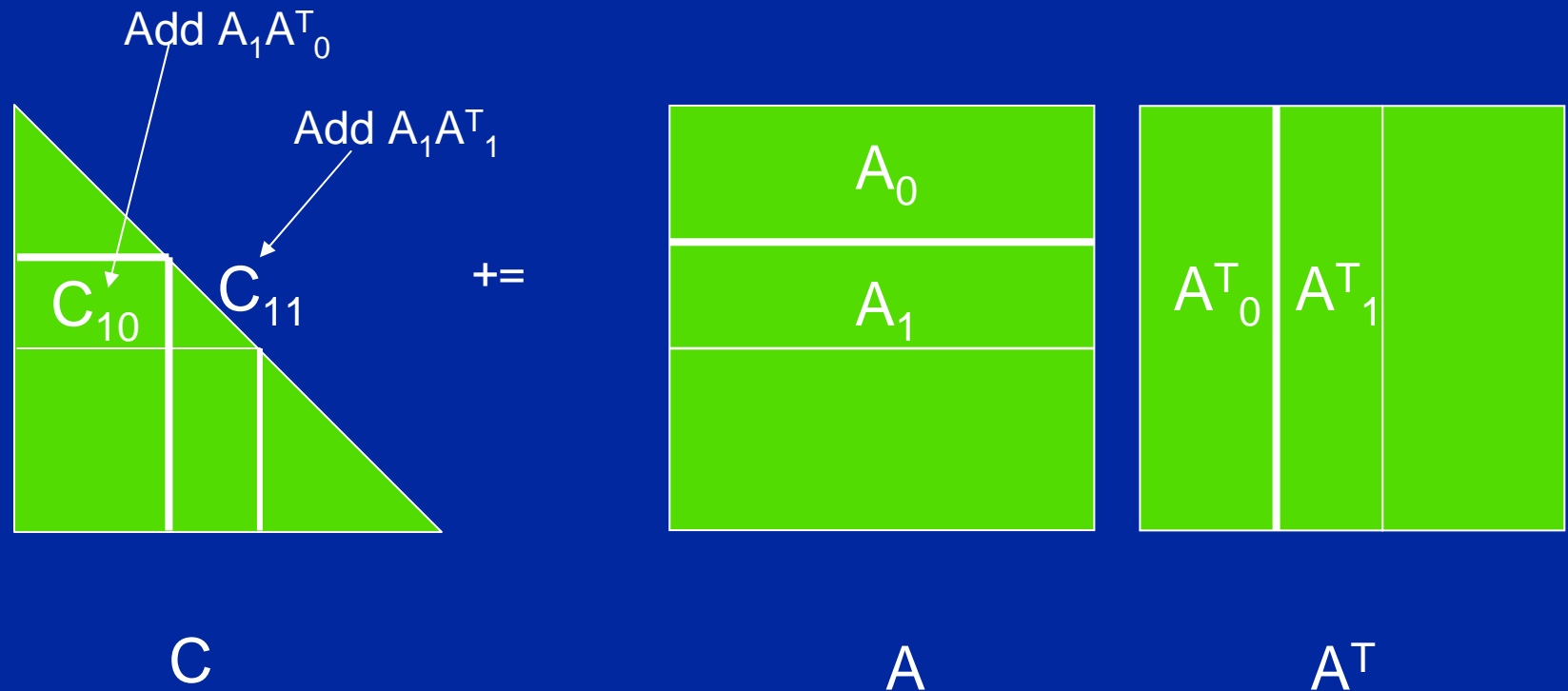
Indices are the root of all evil

The right way to write math libraries

FLAME: www.cs.utexas/users/flame



Symmetric rank-k update



Note: the iteration sweeps through C and A , creating a new block of rows to be updated with new parts of A . These updates are completely independent.


```

while ( FLA_Obj_length( CTL ) < FLA_Obj_length( C ) ){
    b = min( FLA_Obj_length( CBR ), nb_alg );

    FLA_Repart_2x2_to_3x3( CTL, /**/ CTR,    &C00, /**/ &C01, &C02,
                          /***/ /***/
                          &C10, /**/ &C11, &C12,
                          CBL, /**/ CBR,    &C20, /**/ &C21, &C22,
                          b, b, FLA_BR );
    FLA_Repart_2x1_to_3x1( AT,              &A0,
                          /* ** */          /* ** */
                          &A1,
                          AB,              &A2,    b, FLA_BOTTOM );
    /*-----*/

    FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE, ONE, A0, A1, ONE, C10 );
    FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, ONE, A1, ONE, C11 );

    /*-----*/
    FLA_Cont_with_3x3_to_2x2( &CTL, /**/ &CTR,  C00, C01, /**/ C02,
                              C10, C11, /**/ C12,
                              /***/ /***/
                              &CBL, /**/ &CBR,  C20, C21, /**/ C22,
                              FLA_TL );
    FLA_Cont_with_3x1_to_2x1( &AT,              A0,
                              A1,
                              /* ** */          /* ** */
                              &AB,              A2,    FLA_TOP );
}

```

```

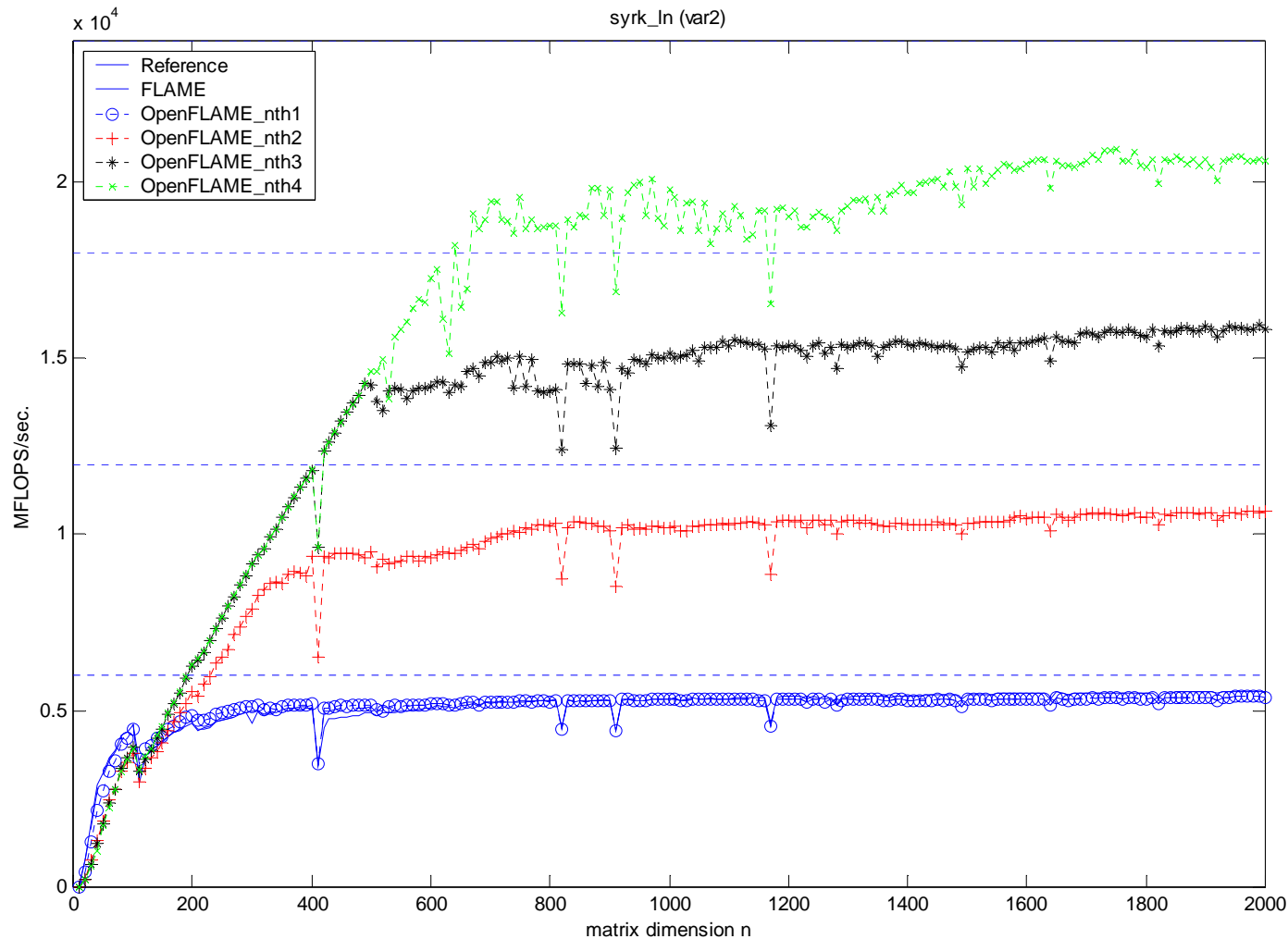
#pragma intel omp parallel taskq
{
    while ( FLA_Obj_length( CTL ) < FLA_Obj_length( C ) ){
        b = min( FLA_Obj_length( CBR ), nb_alg );

        FLA_Repart_2x2_to_3x3( CTL, /**/ CTR,    &C00, /**/ &C01, &C02,
                               /*****/ /*****/
                               &C10, /**/ &C11, &C12,
                               CBL, /**/ CBR,    &C20, /**/ &C21, &C22,
                               b, b, FLA_BR );
        FLA_Repart_2x1_to_3x1( AT,                &A0,
                               /* ** */           /* ** */
                               AB,                &A1,
                                               &A2,    b, FLA_BOTTOM );
        /*-----*/
        #pragma intel omp task captureprivate( A0, A1, C10, C11 )
        {
            FLA_Gemm( FLA_NO_TRANSPOSE, FLA_TRANSPOSE, ONE, A0, A1, ONE, C10 );
            FLA_Syrk( FLA_LOWER_TRIANGULAR, FLA_NO_TRANSPOSE, ONE, A1, ONE, C11 );
        } /* end task */
        /*-----*/
        FLA_Cont_with_3x3_to_2x2( &CTL, /**/ &CTR,  C00, C01, /**/ C02,
                                   C10, C11, /**/ C12,
                                   /*****/ /*****/
                                   &CBL, /**/ &CBR,  C20, C21, /**/ C22,
                                   FLA_TL );

        FLA_Cont_with_3x1_to_2x1( &AT,                A0,
                                   A1,
                                   /* ** */           /* ** */
                                   &AB,                A2,    FLA_TOP );
    }
} /* end of taskq */

```

Top line represents peak of
Machine (Itanium2 1.5GHz, 4CPU)



Note: the above graphs is for the most naïve way of marching through the matrices.
By picking blocks dynamically, much faster ramp-up can be achieved.

So OpenMP is great and getting better

- The current specification (2.5) is very effective for loop oriented programs common in scientific computing.
- The next specification (3.0 ... in Nov'06) will handle a much larger variety of control structures.

But there are major challenges ahead!

- OpenMP was created with a particular abstract machine or *computational model* in mind:
 - Multiple processing elements.
 - A shared address space with “equal-time” access for each processor.
 - Multiple light weight processes (threads) managed outside of OpenMP (the OS or some other “third party”).



But there are major challenges ahead!

- There are no machines on the market today that truly match the OpenMP model:
 - Processors have caches for fast “local” memory.
 - Large multiprocessor machines have NUMA memory subsystems.
 - Multi-core only complicates the problem.
- What are we going to do?
 - We are studying ways to define processor/thread hierarchies in OpenMP ... we hope to get this into OpenMP 3.0.
 - Cluster OpenMP?

Conclusion/Summary

- OpenMP is an important standard for programming shared memory machines.
- OpenMP 3.0 will extend OpenMP to a much larger range of algorithms:
 - OpenFLAME shows the power of incremental parallelism, flexible task queue constructs, and intelligent algorithm design.
- But we have important challenges to resolve
 - How should OpenMP evolve to fit a world where memory hierarchies are complex? NUMA? Clusters?